Rewriting Models of Boolean Programs

Ahmed Bouajjani[1⋆] and Javier Esparza[2⋆⋆]

[1] LIAFA, University of Paris 7, abou@liafa.jussieu.fr
[2] IFMCS, University of Stuttgart esparza@informatik.uni-stuttgart.de

Abstract. We show that rewrite systems can be used to give semantics to imperative programs with boolean variables, a class of programs used in software model-checking as over- or underapproximations of real programs. We study the classes of rewrite systems induced by programs with different features like procedures, concurrency, or dynamic thread creation, and survey a number of results on their word problem and their symbolic reachability problem.

1 Introduction

Software Model Checking is an active research area whose goal is the application of model-checking techniques to the analysis and verification of programs. It devotes a lot of attention to *boolean programs*, which are imperative, possibly nondeterministic programs acting on variables of boolean type. The reason is that boolean programs can be used as over- or underapproximations of the real program one wishes to analyze. In order to obtain underapproximations, one restricts the range of the variables to a small, finite domain. Once this has been done, an instruction of the program can be simulated by an instruction acting on a number of boolean variables, one for each bit needed to represent the finite range (for instance, if we restrict the range of an integer variable v to the interval $[0..3]$ we can simulate an assignment to v by a simultaneous assignment to two boolean variables). The executions of the underapproximation correspond to the executions of the program in which the values of the variables stay within the specified range.

Overapproximations are obtained by *predicate abstraction* [22]. In this approach, one defines a set of boolean predicates on the variables of the program (e.g., $x \leq y$ for two integer variables x and y) and defines an abstraction function that assigns to a valuation of the program variables the set of predicates that it satisfies. Using standard abstract interpretation techniques [15], one can then construct a boolean program having the same control structure as the original one, but now acting on a set of boolean variables, one for each predicate. In Software Model Checking, these approximations are progressively refined in an automatic way until the property is proved, refuted, or until the tools run out of memory. This technique is called *counterexample-guided abstraction refinement* [14].

Boolean while programs have a finite state space, and can be analyzed using standard model-checking techniques. However, modern software goes far beyond while

⋆ Partially supported by the French Ministry of Research ACI project Persée
⋆⋆ Partially supported by the DFG project “Algorithms for Software Model Checking”.

programs: Programs can exhibit recursion, parallelism, and thread creation. Each one of these features leads to an infinite state space, and to questions about the decidability and complexity of analysis problems. In order to attack these questions we need to find semantic models linking boolean programs to formal models with a strong theory and powerful analysis algorithms. This has been the subject of intensive research since the late 90s.

This paper shows that semantic models for boolean programs can be elegantly formulated as rewrite systems. In this approach, program states are formalized as terms, and program instructions as rewrite rules. A step of the program is matched by a rewrite step in its corresponding rewrite system. The nature of the program determines the class of terms we use. In particular, we use string-rewriting and multiset-rewriting as special cases of term rewriting.

Once we have a rewrite model, we wish to analyze it. From the model-checking or program analysis point of view questions like termination and confluence play a minor rôle. One is far more interested in the word problem, and actually on a generalization of it: Given a rewriting system and two (possibly infinite!) sets of terms T and T', can some element of T be rewritten into an element of T'? The software model checking community has attacked this question by studying *symbolic reachability* techniques. In this approach, one tries to find data structures providing finite representations of a sufficiently interesting class of infinite sets of terms, and satisfying at least one of the two following properties: (1) if a set T is representable, then the set $post^*(T)$ of terms reachable from T by means of an arbitrary number of rewriting steps is also representable; moreover, its representation can be effectively computed from the representation of T, and (2) same property with the set $pre^*(T)$ of terms that can be rewritten into terms of T instead of $post^*(T)$.

We survey a number of results on symbolic reachability algorithms for different classes of programs. We start with sequential programs, move to concurrent programs without recursion and, finally, consider the difficult case of concurrent programs with recursive procedures. For each class we give a small example of a program and its semantics, and then present analysis results.

2 Sequential programs

Consider the program of Figure 1. It consists of two procedures, $main()$ and $p()$, and has no variables. The intended semantics of **if** ? **then** c_1 **else** c_2 **fi** is a nondeterministic choice between c_1 and c_2. The program state is not determined by the current value of the program counter only; we also need information about the procedure calls that have not terminated yet. This suggests to represent a state of the program as a *string* $p_0 p_1 \dots p_n$ where p_0 is the current value of the program counter and $p_1 \dots p_n$ is the stack of return addresses of the procedure calls whose execution has not terminated yet. For instance, the initial state of the program of Figure 1 is m_0, but the state reached after the execution of m_1 : **call**$p()$ is not p_0, it is the string $p_0 m_2$.

We can capture the behaviour of the program by the set of *string-rewriting* rules on the right of Figure 1. A procedure call is modelled by a rule of the form $X \to YZ$, where X is the current program point, Y the initial program point of the callee, and

```
procedure p();
p0:  if (?) then
p1:      call main();
p2:      if ? then call p() fi
     else
p3:          call p()
     fi;
p4:  return

procedure main();
m0:  if ? then return fi;
m1:  call p;
m2:  return
```

$$p_0 \to p_1 \quad p_0 \to p_3 \quad p_1 \to m_0\,p_2 \quad p_2 \to p_0\,p_4 \quad p_2 \to p_4 \quad p_3 \to p_0\,p_4 \quad p_4 \to \varepsilon \quad m_0 \to \varepsilon \quad m_0 \to m_1 \quad m_1 \to p_0\,m_2 \quad m_2 \to \varepsilon$$

Fig. 1. A sequential program and its semantics.

Y the return address of the caller. A return instruction is modelled by a rule $X \to \varepsilon$, where ε denotes the empty string. However, with the ordinary rewriting policy of string-rewriting systems

$$\frac{X \to w}{u\,X\,v \xrightarrow{r} u\,w\,v}$$

where $\xrightarrow{r}$ denotes a rewrite step, we have $m_0\,p_2\,m_2 \xrightarrow{r} m_0\,p_0\,p_4\,m_2$ (rule $p_2 \to p_0\,p_4$), which is not allowed by the intuitive semantics. We need to use the *prefix-rewriting policy*

$$\frac{X \to w}{X\,v \xrightarrow{r} w\,v}$$

instead. We also need to interpret ε as the empty string. With these changes we have for instance the rewriting chain

$$m_0 \xrightarrow{r} m_1 \xrightarrow{r} p_0\,m_2 \xrightarrow{r} p_1\,m_2 \xrightarrow{r} m_0\,p_2\,m_2 \xrightarrow{r} p_2\,m_2 \xrightarrow{r} p_4\,m_2 \xrightarrow{r} m_2 \xrightarrow{r} \varepsilon$$

Notice that the string-rewriting system of Figure 1 is *monadic*, i.e., the left-hand-side of the rewrite rules consists of one single symbol.

2.1 Adding variables

Consider the program of Figure 2, where b is a global variable and l is a local variable of the function $foo()$. In the presence of variables, a state of a sequential program can be modelled as a string over the alphabet containing

- a symbol for every valuation of the global variables; and
- a symbol $\langle v, p\rangle$ for every program point p and for every valuation v of the local variables of the procedure p belongs to.

```
bool function foo(l);
f0:  if l then
f1:      return false
     else
f2:      return true
     fi

procedure main();
m0:  while b do
m1:      b := foo(b)
     od
m2: return
```

$$
\begin{aligned}
b\langle t, f_0\rangle &\to b\langle t, f_1\rangle \\
b\langle f, f_0\rangle &\to b\langle f, f_2\rangle \\
b\langle l, f_1\rangle &\to f \\
b\langle l, f_2\rangle &\to t \\
t\, m_0 &\to t\, m_1 \\
f\, m_0 &\to \varepsilon \\
b\, m_1 &\to b, \langle b, f_0\rangle\, m_0 \\
b\, m_2 &\to \varepsilon
\end{aligned}
$$

Fig. 2. A sequential program with global and local variables and its semantics.

States are modelled by strings of the form $g\langle v_1, p_1\rangle \ldots \langle v_n, p_n\rangle$, where g encodes the current values of the global variables, and each pair $\langle v_i, p_i\rangle$ corresponds to a procedure call that has not terminated yet. The symbol v_i encodes the values of the local variables of the caller right before the call takes place, while p_i encodes the return address at which execution must be resumed once the callee terminates. It is straightforward to assign rewrite rules to the program instructions. For instance, the call to $foo(b)$ in $main()$ is modelled by the rules

$$t\, m_1 \to t\,\langle t, f_0\rangle\, m_0 \qquad \text{and} \qquad f\, m_1 \to f\,\langle f, f_0\rangle\, m_0$$

indicating that control is transferred to f_0, that the local variable l gets assigned the current value of the global variable b, and that the return address is m_0. The complete set of rules is shown on the right of Figure 2. The symbols b and l stand in the rules for either **true** or **false**.

Notice that, due to the presence of global variables, the rewrite system is no longer monadic, although the left-hand-sides of the rules are strings of length at most 2.

String-rewriting systems using the prefix-rewriting policy are called *pushdown systems*, due to their similarity with pushdown automata: Given a string $g\langle v_1, p_1\rangle \ldots \langle v_n, p_n\rangle$ modelling a program state, the valuation g of the global variables corresponds to the current control state of the automaton, while the rest of the string corresponds to the current stack content.

2.2 Analysis

String-rewriting systems with prefix-rewriting have an interesting story. They seem to have been studied for the first time by Büchi [9], who called them *canonical systems* (see also Chapter 5 of his unfinished book [10]). Büchi proved the fundamental result that given a regular set S of strings, the sets $pre^*(S)$ and $post^*(S)$ are also regular. The result was rediscovered by Caucal [12]. Book and Otto (who were also unaware of Büchi's work) proved that $pre^*(S)$ is regular for *monadic* string-rewriting systems with *ordinary* rewriting and presented a very simple algorithm that transform a finite

automaton accepting S into another one accepting $pre^*(S)$. This algorithm was adapted to pushdown systems in [2, 21]), and their performance was improved in [18].

Theorem 1. *[2, 21, 18] Given a pushdown system R and a finite-state automaton A, the sets $post^*(L(A))$ and $pre^*(L(A))$ are regular and effectively constructible. in polynomial time in the sizes of R and A.*

More precisely, let P be the set of control states of R, and let Q and δ be the sets of states and transitions of the automaton A, respectively. Let $n = \max\{|Q|, |P|\}$. The automaton representing $post^*(L(A))$ can be constructed in $O(|P||R|(n+|R|)+|P||\delta|)$ time and space, and the automaton representing $pre^*(L(A))$ can be constructed in $O(n^2|R|)$ time and $O(n|R|+|\delta|)$ space.

The theory of pushdown systems and related models (canonical systems, monadic string-rewriting systems, recursive state machines, context-free processes, Basic Process Algebra, etc.) is very rich, and even a succinct summary would exceed the scope of this paper. A good summary of the results up to the year 2000 can be found in [11].

The algorithms of Theorem 1 have found interesting applications. They constitute the core of the Moped tool, Schwoon's back-end for model-checking software, and of its Java front-end jMoped [34]. They are also at the basis of the MOPS tool [13].

3 Concurrent programs without procedures

Programming languages deal with concurrency in many different ways. In scientific computing *cobegin-coend* sections are a popular primitive, while object-oriented languages usually employ *threads*. We consider both variants. Languages also differ in their synchronization and communication mechanisms: shared variables, rendezvous, asynchronous message passing. This point is less relevant for this paper, and we only consider the shared variables paradigm. In this section we consider programs without procedures. the combination of concurrency and procedures is harder to analyze, and we consider it in the next section.

3.1 Threads

The program on the left of Figure 3 spawns a new thread $p()$ each time the while loop of *main*() is executed. This thread runs concurrently with *main*() and with the other instances of $p()$ spawned earlier. Threads communicate with each other through shared variables, in this case the global variable b. Since $p()$ nondeterministically decides whether b should be set to **true** or **false**, *main*() can create an unbounded number of instances of $p()$.

The state of the program can be modelled as a *multiset* containing the following elements:

- the current value of the global variable b,
- the current value of the program counter for the *main*() thread, and
- the current value of the program counter for each thread $p()$.

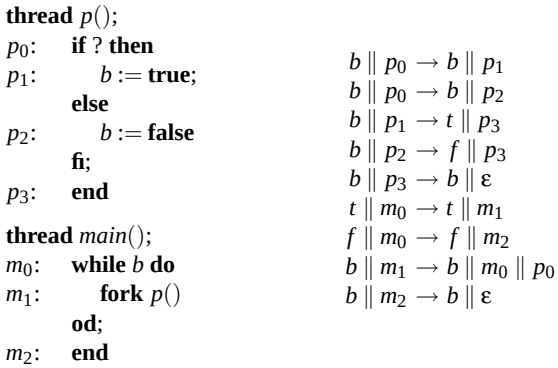

Fig. 3. A program with dynamic thread generation and its semantics.

For instance, the multiset $\{0, m_1, p_1, p_2, p_2\}$ is a possible (and in fact reachable) state of the program with four threads. In order to model the program by means of rewrite rules we introduce a parallel composition operator $\|$ and model the state as $(0 \parallel m_1 \parallel p_1 \parallel p_2 \parallel p_2)$. Intuitively, we consider a global variable as a process running in parallel with the program and communicating with it. We rewrite modulo the equational theory of $\|$, which states that $\|$ is associative, commutative, and has the empty multiset (denoted again by ε) as neutral element:

$$u \parallel (v \parallel w) = (u \parallel v) \parallel w \qquad u \parallel v = v \parallel u \qquad u \parallel \varepsilon = u\,.$$

Observe that, since we rewrite modulo the equational theory, it does not matter which rewriting policy we use (ordinary or prefix-rewriting). The complete set of rewrite rules for the program of Figure 3 is shown on the right of the figure. As in the non-concurrent case, if the program has no global variables then the rewrite system is monadic. Observe that without global variables *no communication between threads is possible*.

Notice that instructions like $p : \mathbf{wait}(b); p' : \ldots$ forcing a thread to wait until the global variable b becomes true can be modelled by the rule $t \parallel p \to t \parallel p'$.

Analysis. While the word problem for pushdown systems can be solved in polynomial time (Theorem 1), it becomes harder for multiset rewriting.

Theorem 2. *[24, 17] The word problem for monadic multiset-rewriting systems is NP-complete.*

NP-hardness can be proved by a straightforward reduction to SAT, while membership in NP requires a little argument. We can also prove a result similar to Theorem 1. In order to formulate the result, observe first that a multiset M over an alphabet $A = \{a_1, \ldots, a_n\}$ can be represented by the vector $\langle x_1, \ldots, x_n \rangle \in \mathbb{N}^n$, where x_i, $i \in \{1, \ldots, n\}$, is the number of occurrences of a_i in M. This encoding allows to represent sets of multisets by means of arithmetical constraints on integer vectors. The sets of vectors definable by formulas of Presburger arithmetic are called *semi-linear sets*. This name is due to the fact that

every semi-linear set is a finite union of *linear sets*, defined as follows. A set $V \subseteq \mathbb{N}^n$ is linear if there is a *root vector* $v_0 \in \mathbb{N}^n$ and a finite number of *periods* $v_1, \ldots, v_k \in \mathbb{N}^n$ such that

$$V = \{v_0 + n_1 v_1 + \ldots, n_k v_k \mid n_1, \ldots, n_k \in \mathbb{N}\}.$$

Semi-linear sets share many properties with regular sets. They are closed under boolean operations. Moreover, if we associate to each word w of a regular language its *Parikh image* (the multiset containing as many copies of each symbol a as there are occurrences of a in w) we get a semi-linear set of multisets[3]. Conversely, every semi-linear set is the Parikh image of some regular language.

Intuitively, the following theorem states that semi-linear sets are to monadic multiset-rewriting what regular sets are to prefix-rewriting (see Theorem 1).

Theorem 3. *[17] Given a monadic multiset-rewriting system and a semi-linear set of states S, the sets $post^*(S)$ and $pre^*(S)$ are semi-linear and effectively constructible.*

Unfortunately, Theorem 3 does not hold for non-monadic multiset-rewriting systems. It is easy to see that these systems are equivalent to (place/transition) Petri nets. In a nutshell, a rewrite rule

$$(X_1 \parallel \ldots \parallel X_n) \;\rightarrow\; (Y_1 \parallel \ldots \parallel Y_m)$$

corresponds to a Petri net transition that takes a token from the places $X_1, \ldots, X_n$ and puts a token on the places $Y_1, \ldots, Y_m$. It is well-known that for Petri nets $post^*(S)$ can be a non semi-linear set of states even when S is a singleton [23].

The word problem for multiset-rewriting systems is equivalent to the reachability problem for Petri nets, and so, using well-known results of net theory we obtain:

Theorem 4. *[29, 25, 26] The word-problem for multiset-rewriting systems is decidable and EXPSPACE-hard.*

The known algorithms for the reachability problem of Petri nets are too complicated for practical use (not to speak of their complexity, which exceeds any primitive-recursive function). However, many program analysis problems can be stated as *control point reachability* problems in which we wish to know if a program point can be reached by a thread, independently of which or how many other threads run in parallel with it. In multiset-rewriting terms, the question is if the rewrite system associated to the program can reach a state of the form $X \parallel t$ for some multiset t. This target set of states is *upward-closed*: if some term t belongs to the set, then $t \parallel t'$ also belongs to the set for every multiset t'. Moreover, multiset-rewriting systems have the following important property: if $t \xrightarrow{r} t'$, then $t \parallel t'' \xrightarrow{r} t' \parallel t''$ for every multiset t''. This makes them *well-structured* systems in the sense of [1, 20], and allows to apply a generic backward reachability algorithm to the control-reachability problem. More precisely, one can show that (1) every upward-closed set admits a finite representation (its set of minimal multisets), (2) if U is upward-closed then $U \cup pre(U)$ is upward-closed, where $pre(U) = \{t \mid \exists u \in U : t \xrightarrow{r} u\}$, and (3) every sequence $U_1 \subseteq U_2 \subseteq U_3 \ldots$ of upward-closed sets reaches a fixpoint after

[3] Parikh's theorem states the same result for context-free languages.

finitely many steps. The generic backwards reachability algorithm iteratively computes (the finite representations of) $U, U \cup pre(U), U \cup pre(U) \cup pre^2(U) \ldots$ until the fixpoint is reached. So we have:

Theorem 5. *Given a multiset-rewriting system and an upward-closed set of states S, the set $pre^*(S)$ is upward-closed and effectively constructible.*

The approach we described above has been adopted for instance in [16] for the verification of multithreaded Java programs.

3.2 Cobegin-coend sections

Another popular way of introducing concurrency is by means of cobegin-coend sections. Intuitively, in the program $(\textbf{cobegin}\, c_1 \parallel c_2\, \textbf{coend})\,; c_3$ the code c_1 and c_2 is executed in parallel, and execution continues with c_3 after *both* c_1 and c_2 have terminated. The fundamental difference with threads is the existence of an implicit synchronization point at the end of the execution of c_1 and c_2.

Modelling the semantics requires to use term rewriting with two operators, one for sequential and another for parallel composition, which we denote by $\cdot$ and $\parallel$, respectively. For instance, if p_1, p_2, p_3 are the control locations associated to c_1, c_2, c_3 in the expression above, then we model the expression by the term $(v \parallel p_1 \parallel p_2) \cdot p_3$, where v is the current valuation of the global variables. Rewriting takes place modulo the equational theory of the $\cdot$ and $\parallel$ operators:

$$\begin{array}{ll} u \cdot (v \cdot w) = (u \cdot v) \cdot w & u \parallel (v \parallel w) = (u \parallel v) \parallel w \\ \varepsilon \cdot u = u & \varepsilon \parallel u = u \\ & u \parallel v = v \parallel u \end{array}$$

We also have to make the rewriting policy precise. Intuitively, it says that we can only rewrite the leftmost part of the syntax tree of a term. Formally,

$$\frac{X \to w}{X \cdot v \xrightarrow{r} w \cdot v} \quad \text{and} \quad \frac{X \to w}{X \parallel v \xrightarrow{r} w \parallel v}\,.$$

This model was introduced by Mayr [30, 31] under the name of *Process Rewrite Systems* (PRS). Figure 4 shows a program and its rewriting semantics as PRS. Notice the rule $b \cdot m_0 \to b \parallel m_0$, which allows to make progress after the execution of the instruction $(b := \neg b \parallel b := \textbf{true})$.

PRSs can also model at least part of the interaction between procedures and concurrency, and therefore we delay their analysis until the next section.

4 Putting procedures and concurrency together

The analysis of programs containing both procedures and concurrency is notoriously difficult. It is easy to show that a two-counter machine can be simulated by a boolean program consisting of two recursive procedures running in parallel and accessing one

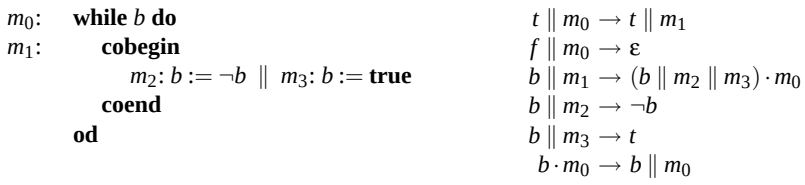

Fig. 4. A program with a cobegin-coend section and its semantics.

single global boolean variable. Intuitively, the two recursion stacks associated to the two procedures are used to simulate the two counters; the depth of the stack corresponds to the current value of the counter. Increments and decrements can be simulated by calls and returns. The global variable is used as a semaphore indicating which counter has to be accessed next. Since two-counter machines are Turing powerful, all interesting analysis problems about these programs are bound to be undecidable.

In programs with procedures and concurrency the same code unit can be called following different policies: procedural call (caller waits until callee terminates), thread call (caller runs concurrently with callee), cobegin-coend call (caller waits, may call several callees). We use the keyword **process** to denote such a unit.

4.1 Procedural programs with cobegin-coend sections

In a while program with cobegin-coend sections the maximum number of processes that can be executed concurrently is syntactically bounded. This is no longer the case in the presence of recursion. For instance, a process may contain a cobegin-coend section one of whose branches calls the process itself. The program of Figure 5 is an example. In the absence of global variables, we can model the program as a monadic PRS (the rules are shown on the right of the figure).

Fig. 5. A procedural program with global variables and its semantics.

Unfortunately, the addition of global variables leads to complications. In order to understand why, consider the program of Figure 6. It is very similar to the program of Figure 5, but has a global variable b. The right side of the figure shows an attempt at

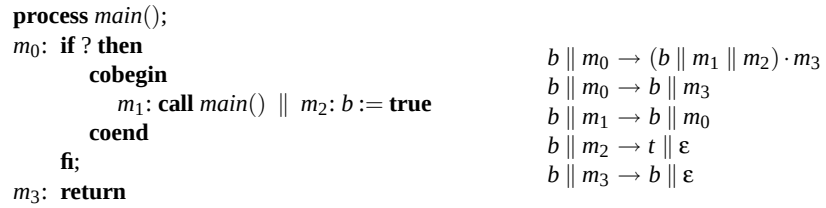

Fig. 6. A program with global variables and an incorrect PRS semantics.

a semantics following the ideas we have used so far. However, the derivations of the rewrite system do not match the intuitive semantics, in which *main*() should be able to call itself, and then execute $b =$ **true** immediately thereafter. No derivation of the rewrite systems allows to do so. The only derivation having a chance would be

$$b\,m_0 \xrightarrow{r} (b \parallel m_1 \parallel m_2) \cdot m_3 \xrightarrow{r} (b \parallel m_0 \parallel m_2) \cdot m_3 \xrightarrow{r} (((b \parallel m_1 \parallel m_2) \cdot m_3) \parallel m_2) \cdot m_3$$

but now the rule $b \parallel m_2 \to b \parallel \varepsilon$ can only be applied to the *innermost* m_2, which corresponds to the incarnation of *main*() as callee, not to its incarnation as caller.

So we conclude that, while monadic PRS are a suitable formalism for modelling programs without global variables, PRS do not match the interplay between recursion and concurrency in conventional programming languages.

Analysis. Mayr has shown that the word problem for PRS is decidable [30, 31] but, since PRS contain Petri nets as a subclass, the problem is EXPSPACE-hard. Fortunately, in the case of monadic PRS (which, as we have seen, seems to be more useful for modelling programs), there exist far more efficient approaches based on symbolic reachability analysis.

The design of symbolic reachability analysis procedures for PRS (or even for its monadic fragment) is not easy. First, we need to represent sets of PRS terms, and a natural idea is to use finite-state tree automata for that. However, we have the problem that the commutative closure of a regular set of terms is not regular. To see this, consider the regular set of terms of the form $a\|b\|a\|b\|\ldots\|a\|b$. Its commutative closure is the set of all parallel terms with the same number of a's and b's, which is clearly not regular. As a consequence, neither the $post^*$ nor the pre^* operation preserves regularity of a set of terms. Moreover, since PRS subsume Petri nets, they do not preserve semi-linearity either.

Nevertheless, in some classes of multiset-rewriting systems, including the monadic class, the $post^*$ operation does preserve semi-linearity, and there exists an algorithm that computes $post^*$ image of any given semi-linear set (see Theorem 3). Let us call a class of multiset-rewriting systems satisfying this property a *semi-linear class*. Given a semi-linear class $\mathcal{C}$, let PRS$[\mathcal{C}]$ be the class of PRS whose sets of rules can be partitioned into two sets M and P, where M is a multiset-rewriting system belonging to the class $\mathcal{C}$, and the rules of P only contain occurrences of the sequential composition operator. Notice that the sets P and M may share constants. We can ask whether

the semi-linearity of $\mathcal{C}$ can be exploited to define an algorithm for symbolic reachability analysis of PRS$[\mathcal{C}]$. This question was addressed in [8] from an automata-theoretic point of view. An important issue is the representation of sets of PRS terms which are closed under the equational theories of $\cdot$ and $\|$. Since these operators are associative, PRS terms can be seen as trees with unbounded width. Each node labelled with $\|$ (resp. $\cdot$) may have an arbitrary number of children labelled with simple symbols (process constants) and an arbitrary number of children labelled with $\cdot$ (resp. $\|$). Therefore, a natural idea is to use *unranked tree automata* (also called *hedge automata*) as symbolic representations for sets of PRS terms. Furthermore, since parallel composition is commutative, we should use *commutative hedge automata (CHA)*. CHA are closed under boolean operations and have a decidable emptiness problem (see also [27, 33]). Then, we have the following generic result:

Theorem 6. *[8] Let $\mathcal{C}$ be a semi-linear class of multiset rewrite systems. For every system in PRS$[\mathcal{C}]$ and for every CHA-definable set of terms T, the sets $post^*(T)$ and $pre^*(T)$ are CHA-definable and effectively constructible.*

By Theorem 3 we know that the class of monadic multiset rewrite systems is semi-linear. Therefore, Theorem 6 gives a procedure for symbolic reachability analysis for monadic PRS:

Theorem 7. *[8] Given a monadic PRS, for every CHA-definable set of terms T, the sets $post^*(T)$ and $pre^*(T)$ are CHA-definable and effectively constructible.*

Actually, using the same approach, it is possible to extend Theorem 7 to a larger subclass of PRS whose rules contain no occurrence of the parallel operator on the left-hand-side (but possibly occurrences of $\cdot$) [8]. This class is called PAD in the literature.

Another approach to the symbolic reachability problem for monadic PRS constructs not the sets $post^*(T)$ or $pre^*(T)$ themselves, but a set of *representatives* w.r.t. the equational theories of sequential and parallel composition. This is sufficient to solve reachability problems where the origin and target sets of terms are closed modulo these equational theories. In particular, the approach is powerful enough to solve control point reachability problems. The approach was first studied in [28] and later in [19]. We rephrase here the result of [19]. Notice that, by essentially the same procedure used to put a context-free grammar in Chomsky normal form, we can transform a monadic PRS into a normal form in which the right-hand-side of all rules has the shape $X \cdot Y$ or $X \| Y$.

Theorem 8. *[28, 19] Let R be a monadic PRS in normal form, and let A be a bottom-up tree automaton recognizing a set $L(A)$ of PRS terms. One can construct in $O(|R| \cdot |A|)$ time two bottom-up tree automata recognizing for every term $t \in post^*(L(A))$ ($t \in pre^*(L(A))$) a term t' such that $t = t'$ in the equational theory of the $\|$ and $\cdot$ operators.*

This approach was extended to the case of PAD systems in [7].

4.2 Multithreaded procedural programs

Process Rewrite Systems are also inadequate for modelling the combination of multithreading and procedures, even in the absence of variables. Consider the program of

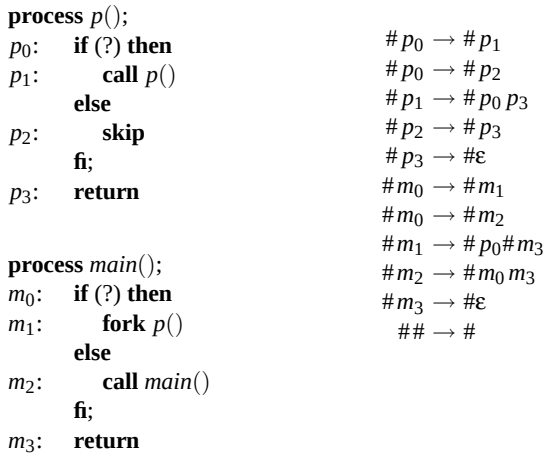

Fig. 7. A program with dynamic thread generation and its semantics.

Figure 7. If we model the **fork** operation by a rule like $m_1 \to m_3 \parallel p_0$, we get the derivation

$$m_0 \xrightarrow{r} m_2 \xrightarrow{r} m_0 \cdot m_3 \xrightarrow{r} m_1 \cdot m_3 \xrightarrow{r} (m_3 \parallel p_0) \cdot m_3 \xrightarrow{r} p_0 \cdot m_3$$

But this is not the intended semantics. The main thread (corresponding to m_3 in the term $p_0 \cdot m_3$) can only terminate *after* the new thread (corresponding to p_0 has terminated.

A new approach has been proposed by the first author, Müller-Olm and Touili in [6]. The idea is to represent a state at which n-threads are active by a string $\#w_n\#w_{n-1}\ldots\#w_1$. Here, $w_1,\ldots w_n$ are the strings modelling the states of the threads, and they are ordered according to the following criterion: for every $1 \leq i < j \leq n$, the i-th thread (i.e., the thread in state w_i) must have been created no later than the j-th thread. The reason for putting younger threads *to the left* of older ones will be clear in a moment.

We can now try to capture the semantics of the program by string-rewriting rules. Notice however that we cannot use the prefix-rewriting policy. Loosely speaking, a thread in the middle of the string should also be able to make a move, and this amounts to rewriting "in the middle", and not only "on the left"'. So we must go back to the ordinary rewriting policy

$$\frac{X \to w}{uXv \xrightarrow{r} uwv}$$

Instructions not involving thread creation are modelled as in the non-concurrent case, with one difference: Since we can only rewrite on the left of a w_i substring, we "anchor" the rewrite rules, and use for instance $\#p_1 \to \#p_0\,p_3$ instead of $p_1 \to p_0\,p_3$. The thread creation at program point m_1 is modelled by the rule $\#m_1 \to \#p_0\#m_3$. Notice that we would not be able to give a rule if we wanted to place the new thread to the right of its creator, because the stack length of the creator at the point of creating the new

thread can be arbitrarily large. This class of string-rewriting systems is called *dynamic networks of pushdown systems* (DPN) in [6]. The complete set of rewrite rules for the program of Figure 7 is shown on the right of the same figure.

Analysis. Notice that DPNs are neither prefix-rewriting nor monadic. However, we still have good analizability results. First of all, it can be proved that the pre^* operation preserves regularity:

Theorem 9. *[6] For every regular set S of states of a DPN, the set $pre^*(S)$ is regular and a finite-state automaton recognizing it can be effectively constructed in polynomial time.*

The $post^*$ operation, however, does not preserve regularity. To see this, consider a program which repeatedly creates new threads and counts (using its stack) the number of threads it has created. The set of reachable states is not regular, because in each of them the number of spawned threads must be equal to the length of the stack. Nevertheless, the $post^*$ operation preserves context-freeness.

Theorem 10. *[6] For every context-free (pushdown automata definable) set S of states of a DPN, the set $post^*(S)$ is context-free and a pushdown automaton recognizing it can be effectively constructed in polynomial time.*

Since intersection of a regular language with a context-free language is always context-free, and since the emptiness problem of context-free languages is decidable, this result allows to solve the reachability problem between a context-free initial set of configurations and a regular set of target configurations.

So far we have only considered the variable-free case. The results above can be extended to the case in which processes have local variables, but global variables make the model Turing powerful. In this case over/underapproximate analysis approaches can be adopted, which are outside the scope of this paper (see, e.g., [4, 5, 32, 3]).

5 Summary

We have studied rewriting models for sequential and concurrent boolean programs where concurrent processes communicate through shared variables.

Sequential boolean programs with procedure calls can be modelled by prefix-rewriting systems (pushdown systems). The word problem and the symbolic reachability problem for regular sets of states can be solved in polynomial time. The algorithms have been implemented in the Moped and MOPS tools and applied to the analysis of large programs.

Concurrent programs with dynamic thread creation but without procedures can be modelled by multiset-rewriting systems (Petri nets). The word problem is decidable, but the algorithm is not useful in practice. The control reachability problem can be solved by a simple algorithm based on the theory of well-structured systems. The symbolic reachability problem can be solved for the monadic fragment and semi-linear sets. The

monadic fragment corresponds to programs without global variables, and so to absence of communication between threads.

Process Rewrite Systems (PRS) combine prefix-rewriting and multiset-rewriting. PRS have a decidable word problem, but do not match the interplay between procedures and concurrency in conventional programming languages. Monadic PRS model parallel programs with cobegin-coend sections and procedure calls, but without global variables. The word problem is NP-complete. The control reachability problem can be solved very efficiently using bottom-up tree automata. The symbolic reachability problem can be solved for sets of states recognizable by commutative hedge automata.

Concurrent programs with thread creation and procedures, but without communication between threads, can be model by dynamic networks of pushdown systems [6], a class of string-rewriting systems. The word problem can be solved in polynomial time. The pre^* operation preserves regularity (and can be computed in polynomial time), while the $post^*$ operation preserves context-freeness.

Concurrent programs with procedures and one single global variable are already Turing powerful, and so very difficult to analyze. Several approximate analysis have been proposed based on the automata techniques presented in this paper (see e.g. [4, 5, 32, 3]). The constrained dynamic networks of [6] replace global variables by a more restricted form of communication in which a process can wait for a condition on the threads it created, or for a result computed by a procedure it called.

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.
3. A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS*, volume 3821 of *Lecture Notes in Computer Science*. Springer, 2005.
4. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *Int. J. Found. Comput. Sci.*, 14(4):551–, 2003.
5. A. Bouajjani, J. Esparza, and T. Touili. Reachability analysis of synchronized pa systems. *Electr. Notes Theor. Comput. Sci.*, 138(3):153–178, 2005.
6. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*. Springer, 2005.
7. A. Bouajjani and T. Touili. Reachability analysis of process rewrite systems. In *FSTTCS*, volume 2914 of *Lecture Notes in Computer Science*, pages 74–87. Springer, 2003.
8. A. Bouajjani and T. Touili. On computing reachability sets of process rewrite systems. In *RTA*, volume 3467 of *Lecture Notes in Computer Science*. Springer, 2005.
9. J. R. Büchi. Regular canonical systems. *Arch. Math. Logik Grundlag.*, 6:91–111, 1964.
10. J. R. Büchi. *The collected works of J. Richard Büchi.* Springer-Verlag, New-York, 1990.
11. O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on Infinite Structures. In *Handbook of Process Algebra*. North-Holland Elsevier, 2001.
12. D. Caucal. On the regular structure of prefix rewriting. *Theor. Comput. Sci.*, 106(1):61–86, 1992.

13. H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244, 2002.
14. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
15. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
16. G. Delzanno, J.-F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded java programs. In *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2002.
17. J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundam. Inform.*, 31(1):13–25, 1997.
18. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.
19. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL*, pages 1–11, 2000.
20. A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
21. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electr. Notes Theor. Comput. Sci.*, 9, 1997.
22. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.
23. J.E. Hopcroft and J.-J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theor. Comput. Sci.*, 8:135–159, 1979.
24. D.T. Huynh. Commutative grammars: The complexity of uniform word problems. *Information and Control*, 57(1):21–39, 1983.
25. S. R. Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *STOC*, pages 267–281. ACM, 1982.
26. R. Lipton. The Reachability Problem Requires Exponential Space. Technical Report 62, Yale University, 1976.
27. D. Lugiez. Counting and equality constraints for multitree automata. In *FoSSaCS*, volume 2620 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2003.
28. D. Lugiez and Ph. Schnoebelen. The regular viewpoint on PA-processes. In *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1998.
29. E.W. Mayr. An algorithm for the general Petri net reachability problem. In *STOC*, pages 238–246. ACM, 1981.
30. R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, Technische Universität München, 1998.
31. R. Mayr. Process rewrite systems. *Inf. Comput.*, 156(1-2):264–286, 2000.
32. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.
33. H. Seidl, Th. Schwentick, and A. Muscholl. Numerical Document Queries. In *PODS'03*. ACM press, 2003.
34. D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 541–545, Edinburgh, UK, 2005. Springer.